Parallel Implementation of Divide-and-Conquer Semiempirical Quantum Chemistry Calculations

WEI PAN,[1] TAI-SUNG LEE,[1]* WEITAO YANG[2]†‡

[1]*Department of Chemistry, Paul M. Gross Chemical Laboratory, Duke University, Box 90354, Durham, North Carolina 27708-0354*

[2]*Department of Chemistry, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong*

Received 24 July 1997; accepted 23 February 1998

ABSTRACT: We have implemented a parallel divide-and-conquer method for semiempirical quantum mechanical calculations. The standard message passing library, the message passing interface (MPI), was used. In this parallel version, the memory needed to store the Fock and density matrix elements is distributed among the processors. This memory distribution solves the problem of demanding requirement of memory for very large molecules. While the parallel calculation for construction of matrix elements is straightforward, the parallel calculation of Fock matrix diagonalization is achieved via the divide-and-conquer method. Geometry optimization is also implemented with parallel gradient calculations. The code has been tested on a Cray T3E parallel computer, and impressive speedup of calculations has been achieved. Our results indicate that the divide-and-conquer method is efficient for parallel implementation. © 1998 John Wiley & Sons, Inc. J Comput Chem 19: 1101–1109, 1998

Keywords: parallel computation; divide-and-conquer; semiempirical; quantum mechanical

* *Present address:* Department of Pharmaceutical Chemistry, University of California, San Francisco, California

† A. P. Sloan Fellow

‡ On leave from the Department of Chemistry, Duke University, Durham, North Carolina

Correspondence to: W. Yang

Contract/grant sponsors: U.S. Environmental Protection Agency; National Science Foundation; National Institutes of Health

© 1998 John Wiley & Sons, Inc.
CCC 0192-8651 / 98 / 091101-09

Introduction

Quantum mechanical description of electronic structure is very important for many aspects of molecular modeling. However, it is still a great challenge to study large and complex molecular systems. Computation of large systems is very expensive because conventional quantum chemistry methods exhibit a quadratic or higher scaling of computational requirements with the size of the system simulated. The successful approach to large systems requires a combination of innovation in theoretical and computational methods and an increase in computer power.

In the theoretical aspect, much effort has been made in the development of linear scaling quantum calculations[1–9]; that is, methods that require computational effort proportional to the size of the system. The bottleneck of quantum chemistry calculations originates mainly from two situations, the N^2 scaling of the construction of Fock matrix elements and the N^3 scaling of the diagonalization process of the matrix. In the first situation, semiempirical quantum mechanical methods provide a very efficient way to construct the Fock matrix elements from their approximations. The divide-and-conquer linear-scaling algorithm has been demonstrated to circumvent the N^3 scaling of diagonalization problem.[1,3] In the recent implementation of this approach, semiempirical quantum mechanical calculations were made possible for large molecules of over 9000 atoms.[10] The divide-and-conquer algorithm has also been applied to semiempirical quantum mechanical calculations by Merz et al.[11] Stewart developed a promising localized molecular orbitals method based on partial diagonalization.[12] Instead of diagonalization, Stewart's method transforms the Hamiltonian matrix into block diagonal form, with zero values for the elements connecting the occupied and unoccupied orbitals. Scuseria et al. applied the density matrix method[5] to semiempirical approaches.[13]

In regard to computing power, parallel computers have opened the door to the study of increasingly large molecular systems.[14–18] Parallel computing, in which an array of processors is used collectively to solve a single large problem, is emerging as a major tool useful in many scientific fields, including computational chemistry. In particular, Thiel et al. have implemented semiempirical quantum chemistry methods in distributed memory parallel computers.

Combining parallel computation with the current linear-scaling quantum mechanical theoretical methods can potentially provide very powerful approaches for the study of large molecular systems. In the divide-and-conquer approach,[1] linear scaling is achieved by dividing a large molecular system into subsystems. Instead of diagonalizing the Fock matrix of whole system, many relatively small-sized Fock matrices of the subsystems are diagonalized. It is important to note that the diagonalizations of each of the subsystems are independent computing tasks, which provides a natural basis for taking full advantage of parallel computers.

A major impediment to widespread use of parallel machines has been the difficulty in implementing traditional serial programs on parallel architectures. Recent advances in programming language, compilers, and other system software have made the transition a little easier. There are several general paradigms for implementation of parallel programs. The message passing model, in which each processor of the machine is able to execute independently of other processors, seems suitable for our purpose. Among the number of packages that allow for portable message passing coding, we have chosen MPI[19] as our parallel programming environment, although our implementation can be easily ported to other packages that support a message passing paradigm.

In this article, we describe the parallel implementation of semiempirical divide-and-conquer quantum mechanical calculations. We first briefly review the basic method for solution of the problem. We then describe our parallel model with special emphasis on memory distribution, construction of the Fock matrix, and the diagonalization process. In the final section, we present the performance of our code for several representative examples, and a discussion of the observed performance.

Theory

In the density matrix formalism of the divide-and-conquer approach,[1,3] the density matrix is divided into subsystem contributions by the use of the symmetric weight matrices, P^α:

$$\rho_{ij} = \sum_\alpha P_{ij}^\alpha \rho_{ij} \equiv \sum_\alpha \rho_{ij}^\alpha \tag{1}$$

$$\sum_\alpha P_{ij}^\alpha = 1, \forall i, j = 1 \ldots M \tag{2}$$

where α is the index of subsystems and M is the size of the basis sets. Currently, the Mulliken-type weight matrix[20] is used:

$$\begin{aligned} P_{ij}^{\alpha} &= 1 \quad \text{if } i \in \alpha \text{ and } j \in \alpha \\ &= 1/2 \text{ if } i \in \alpha \text{ and } j \notin \alpha \\ &= 0 \quad \text{if } i \notin \alpha \text{ and } j \notin \alpha \end{aligned} \tag{3}$$

Because of the local nature of the density matrix in real space, the density matrix projected into each subsystem can be approximated by solving the expansion coefficients locally; namely:

$$\rho_{ij}^{\alpha} \cong P_{ij}^{\alpha} \sum_{m} n_m^{\alpha} C_{im}^{\alpha} C_{jm}^{\alpha} \tag{4}$$

where n_m^{α} and C_{im}^{α} are the occupation number and the eigenvector of the mth molecular orbital in the α subsystem, respectively. The local eigenvectors for the α subsystem are determined by the subsystem eigenvalue equation:

$$\underline{\underline{H}}^{\alpha} \underline{C}_m^{\alpha} = \underline{\underline{S}}^{\alpha} \underline{C}_m^{\alpha} \epsilon_m^{\alpha} \tag{5}$$

where H^{α} is the molecular one-electron Hamiltonian matrix, S^{α} is the overlap matrix, and $\{\epsilon_m^{\alpha}\}$ are the eigenvalues for the α subsystem.

The occupation number n_m^{α} is approximated by the Fermi function $f_{\beta}(\mu - \epsilon_m^{\alpha})$, with $f_{\beta}(x) = [1 + \exp(-\beta x)]^{-1}$, where β is the inverse temperature, μ is the chemical potential. μ is chosen so that normalization of the density is maintained:

$$N = \sum_{\alpha} \sum_{ij} \rho_{ij}^{\alpha} S_{ij}^{\alpha} \tag{6}$$

where:

$$\rho_{ij}^{\alpha} = P_{ij}^{\alpha} \sum_{m} n_m^{\alpha} C_{im}^{\alpha} C_{jm}^{\alpha} \cong 2 P_{ij}^{\alpha} \sum_{m} f_{\beta}(\mu - \epsilon_m^{\alpha}) C_{im}^{\alpha} C_{jm}^{\alpha} \tag{7}$$

where N is the total number of electrons and the factor 2 accounts for the spin degrees of freedom. In semiempirical methods, the electronic energy is expressed by:

$$E = \frac{1}{2} \sum_{\alpha} \sum_{ij} \rho_{ij}^{\alpha} (H_{ij}^{core} + F_{ij}) \tag{8}$$

where H^{core} is the one-electron core Hamiltonian matrix and F is the Fock matrix. In our calculations, the PM3[21,22] Hamiltonian is used. The energy gradient expressions for the divide-and-conquer approach have been derived and shown to be accurate.[3,23] In semiempirical method, the energy gradients are calculated with the frozen density approximation.[24] With this approximation, the divide-and-conquer energy gradient with respect to the αth nucleus position R is expressed by:

$$\nabla_{R_{\alpha}} E = \frac{1}{2} \sum_{\alpha} \sum_{ij} \rho_{ij}^{\alpha} \nabla_{R_{\alpha}} (H_{ij}^{core} + F_{ij}) \tag{9}$$

Parallel Model

The outline of our parallel model is shown briefly in what follows. The SCF iterations are carried out between STEP3 and STEP7. We focus our description, in detail, on STEP2 to STEP4.

STEP1: read in molecule data
STEP2: assign matrix elements to each processor
allocate memory on each processor to store the matrix elements assigned
STEP3: loop over processors of i
calculate Fock matrix elements in processor i
end loop
STEP4: loop over batches of j
assign each processor with one subsystem
loop over processors of i
get the Fock matrix elements of the subsystem assigned to this processor
(locally communicate with some other processors)
diagonalize the subsystem Fock matrix
end loop
end loop
STEP5: determine chemical potential
STEP6: loop over batches of j
assign each processor with one subsystem
loop over processors of i
calculate density matrix of the subsystem assigned to this processor
send the density matrix elements to those processors assigned to store them
(locally communicate with some other processors)
end loop
end loop
STEP7: calculate total energy
STEP8: calculate energy gradients

In quantum calculations of large molecules, saving memory is as important as reducing execution time. The $O(N^2)$ scaling of the computer memory for storing the Fock and density matrix needs to be addressed first. Because most of the matrix elements in quantum calculations are negligibly small for large molecules, sparse matrix storage methods can be employed. For the density matrix, because of its locality in real space, we can truncate the matrix elements using a distance criterion. Only the matrix elements corresponding to atom pairs within an interatomic distance less than a critical radius R_{cutoff} are evaluated and stored. This cutoff makes the memory storage of the density matrix proportional to the size of the molecules and also reduces the CPU time used for matrix element evaluation. The one-electron core Hamiltonian and Fock matrices are treated similarly. This sparse matrix storage method, used efficiently in the previous sequential code,[10] is kept in the current parallel code. However, with the use of parallel computers, the demanding requirement of memory can be further reduced by distributing the memory over all processors.

MATRIX CONSTRUCTION

In current parallel code (see STEP2), the matrices including the density matrix D, the Fock matrix F, and the one-electron core Hamiltonian matrix H, are distributed over all available processors. We represent the matrix blocks as $M_{\alpha\beta}$, where indices α and β are a collection of indices of orbitals that belong to atom α and β. Index α runs over all atoms in the molecule, and index β will range from 1 to α because of the symmetry of these matrices. We assign an equal number of atoms, on average, to each processor. Each processor stores the blocks of matrix $M_{\alpha\beta}$, where α runs over the atoms assigned to the processor, and β runs over from the first atoms to the αth atom. Obviously, in this way, the size or the number of matrix elements stored in each processor will not be the same. However, this storage imbalance in processors is only moderate for large molecules, because all matrices involved are sparse. Only when atom β is within the cutoff distance R_{cutoff} of atom α, the matrix block $M_{\alpha\beta}$ is stored; otherwise, it is set to be zero and not stored. To further reduce interprocessor communication (at the expense of memory requirement), we also keep, in every processor, a copy of all diagonal blocks of the density matrix $\{D_{\alpha\alpha}$, α runs over all atoms in the molecule$\}$.

With the aforementioned memory distribution scheme, the parallel calculations for construction of matrices F and H are straightforward (see STEP3). Each processor provides the calculations for the F and H matrix elements that are stored in this processor. Because all data (such as $D_{\alpha\beta}$ with α in this processor, and all $D_{\alpha\alpha}$) needed to construct matrices F and H can be found within this processor, communication between processors is not necessary. Each processor constructs the matrix blocks independently. It is important to note that the CPU time for construction of diagonal matrix elements is different from that of off-diagonal matrix elements. The calculation for the former involves the Coulomb interaction, in which the cutoff cannot be used because of its long-range behavior. In this case, the summation loop for all atoms in the system is needed. For off-diagonal matrix elements, we do not need the loop over all atoms because all three- and four-center integrals are completely ignored in the semiempirical approximation. Therefore, the CPU time for diagonal matrix elements will be dominant for large molecules. Because in each processor we have, on average, the same number of diagonal matrix elements, the computing task is very well balanced in our scheme of memory and computation distribution.

MATRIX DIAGONALIZATION

Interprocessor Communication

The parallel diagonalization of the Fock matrix is achieved by employing the linear-scaling divide-and-conquer approach. As we have described [see eq. (5)], the computing task of diagonalizing the whole Fock matrix can be reformulated to a number of computing tasks for diagonalizing small-sized Fock matrices of divided subsystems. In this code (see STEP4), we group the subsystems into batches. Each batch contains the same number of subsystems, which equals the number of processors available. The last batch may have fewer subsystems, because, in some cases, the number of subsystems is not a multiple of the number of processors. Each processor simultaneously gets one subsystem Fock matrix at a time and performs the calculation of diagonalization. When every processor finishes its calculations, the next batch will be assigned.

When one processor needs a subsystem matrix for diagonalization, the processor needs to acquire some matrix elements from other processors, be-

cause not all matrix elements are stored in this processor. In this process, communication between processors is necessary. This kind of communication is also needed in STEP6, after the construction of the matrix via eq. (7), and when we send some of the calculated subsystem density matrix elements back to the processors that are assigned to store them. Communication is unavoidable in all memory distribution methods. However, in our case, the communication should be ''local''—we do not need communication with all processors to obtain the subsystem matrix elements. Particularly when we deal with very large molecular systems, we can divide the system into many subsystems of much smaller size. To obtain the subsystem matrix, only a few processors need to be in communication with one another.

Load Balance

Because the sizes of matrices for the different subsystems are not the same, there are several kinds of load imbalances of diagonalization for the processors. In our implementation, we alleviate this load imbalance by sorting the subsystems according to their sizes. The result of sorting reduces the changes of the subsystem matrix size within a batch to a minimum. Again, when we deal with very large molecular systems, we have many subsystems to be grouped. In this case, we can reasonably predict that the load imbalance due to the different sizes of subsystem matrices within a batch will be small. This kind of imbalance can also be solved, if possible, by manual adjustment of the subsystem sizes when we divide them.

GRADIENT CALCULATIONS

After SCF energy calculation, we can calculate the energy gradients with respect to the nucleus position, as defined in eq. (9). The gradients can be calculated by the finite difference method.[24] With the construction of H^{core} and F matrices being well parallelized in our implementation, the parallel calculation of gradients (STEP8) simply follows that of matrix construction.

MEMORY REQUIREMENT

In our current implementation, all elements of the Fock and the density matrices are approximately evenly distributed, except for the diagonal elements of the density matrix, which are stored in each processor. Even for molecules with 10,000 atoms, the storage space for the diagonal blocks of the density matrix is still very small ($\sim$ 250 kB). Although we could not find suitable tools to measure the actual memory used in calculation for each processor, we believe that the memory required for each processor is proportional to the reciprocal of the number of processors. We found that the minimum number of processors needed to handle HIV protease 8-mer (12,502 atoms) is 12 on a T3E with 128 MB for each PE. Thus, we expect that a 32-processor T3E with 128 MB for each PE should be able to treat molecules with sizes of $\sim$ 30,000 atoms.

OTHER TECHNICAL DETAILS

In this study, we use the same parameters as in the previous sequential code[10]; that is, the matrix cutoff radius R_{cutoff} is set to 7° and the radius of the buffer is 6°. Each residue is defined as a subsystem for proteins and each nucleotide unit is defined as a subsystem for DNA molecules. The geometries used are taken from our previous study.[10]

Performance

We have chosen several representative molecular systems to present the performance of our parallel code. Calculations are carried out on a Cray T3E parallel machine with 128 MB of memory for each processor. The CPU time (seconds per SCF iterations) and speedup data for these systems are listed in Tables I and II and displayed in Figures 1 and 2.

We choose polyglycine chains as our first example. We use the 400-polyglycine molecule (2803 atoms), and divide the molecule into 400 subsystems. Because the polyglycine chain is a typically linear molecule, the communications in our code for this case should be very local and, therefore, require very little time. Furthermore, the sizes of the subsystems are all the same except for those located at the ends of the chain, so the load imbalance for diagonalizing subsystem matrices will also be very small. This kind of linear molecule is the best scenario for the performance of our code. The CPU time of one SCF iteration for a 400-polyglycine molecule is about 213 seconds using one processor, which is reduced dramatically to only 7 seconds if 32 processors are used. The speedup factor achieved is 31.3. In Figure 1, one can see that our speedup data for this molecule are

TABLE I.
CPU Time (Seconds per SCF Iteration) and Speedup versus Number of Processor Elements (PEs).

	Gly-400[a]		DNA[b]		HIV1[c]		HIV2[d]		
PEs	CPU	Speedup	CPU	Speedup	CPU	Speedup	CPU	Speedup	Perfect
1	213.74		669.57		336.30		936.29		
4	53.83	4.0	171.14	3.9	91.79	3.7	244.92	3.8	4
8	27.04	7.9	88.01	7.6	50.29	6.7	128.33	7.3	8
12	18.07	11.8	58.48	11.4	35.69	9.4	87.85	10.7	10
16	13.56	15.7	43.70	15.3	28.55	11.8	68.30	13.7	16
20	11.07	19.3	37.27	18.0	23.32	14.4	55.20	17.0	20
24	9.18	23.3	28.86	23.2	21.13	15.9	47.97	19.5	24
28	7.82	27.3	27.81	24.1	18.99	17.7	42.23	22.2	28
32	6.81	31.3	22.88	29.3	17.70	19.0	37.84	24.7	32

[a]400-polyglycine with 2803 atoms and 400 subsystems.
[b]A 48 −C-G-pair A-DNA with 3024 atoms and 96 subsystems.
[c]HIV protease with 1563 atoms and 99 subsystems.
[d]HIV protease dimer with 3126 atoms and 198 subsystems.

very close to the perfect theoretical speedup for up to 32 processors, which indicates the excellent performance of our code.

For the second example, we choose an A-DNA molecule with 48 C−G pairs. This molecule has 3024 atoms, and is divided into 96 subsystems. The structure of this DNA molecule is a double helix, which is also a nearly linear structure with the size of one dimension being much larger than the size of the other two dimensions. The communication time can then be predicted to be as brief as in the similar case of the 400-polyglycine chain. Except for both ends of this molecule, the size of the subsystems does not change very much, with a size fluctuation of within 20%. The size of this molecule is nearly the same as 400-polyglycine molecule, but we use fewer subsystems. We then examine the effect of using fewer subsystems on the load imbalance of our code (which reduces speedup). The speedup data are shown in Figure 1. Most points for this molecule nearly coincide with those of the 400-polyglycine molecule, except points with 20, 28, and 32 processors. Because 96 is not a multiple of 20 and 28, some processors will remain idle in the last batch job. The relatively large decrease in speedup at these two points is clearly the result of poor load balance in the last batch. Furthermore, we only have five

TABLE II.
CPU Time (Seconds per SCF Iteration) and Speedup versus Number of Processor Elements (PEs).

	HIV4[a]			HIV6[b]			HIV8[c]		
PEs	CPU	Speedup	Perfect	CPU	Speedup	Perfect	CPU	Speedup	Perfect
2	1047.44								
4	523.51	2.0	2						
8	268.71	3.9	4	511.22					
12	182.19	5.7	6	344.32	1.5	1.5			
16	141.16	7.4	8	262.51	2.0	2	405.50		
20	114.56	9.1	10						
24	96.08	10.9	12	179.56	2.9	3	275.70	1.5	1.5
28	84.75	12.4	14						
32	75.49	13.9	16	139.27	3.7	4	212.10	1.9	2

[a]HIV protease 4-mer with 6252 atoms and 396 subsystems.
[b]HIV protease 6-mer with 9378 atoms and 594 subsystems.
[c]HIV protease 8-mer with 12,504 atoms and 792 subsystems.

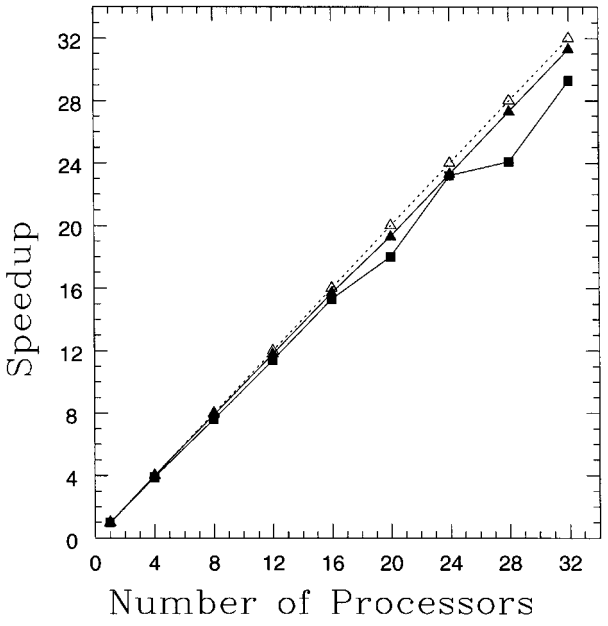

FIGURE 1. Speedup versus the number of processors. Open triangle: the perfect speedup; solid triangle: our speedup for the 400-polyglycine molecule (2803 atoms and 400 subsystems); solid square: our speedup for 48-C −G-pair A-DNA molecule (3024 atoms and 96 subsystems). The calculations were done on a Cray T3E.

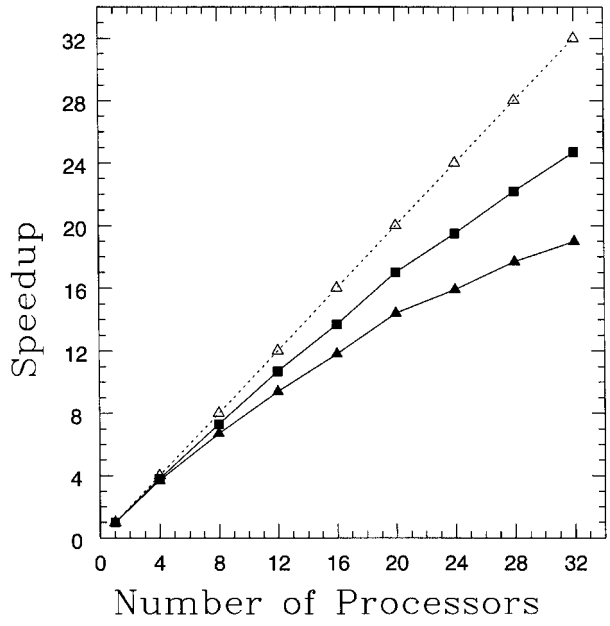

FIGURE 2. Speedup versus the number of processors. Open triangle: the perfect speedup; solid triangle and square: our speedup for protein molecule HIV protease (1563 atoms and 99 subsystems) and HIV protease dimer (3126 atoms and 198 subsystems). The calculations were done on a Cray T3E.

and four batches for 20 and 28 processors used, respectively, and one poorly balanced batch will have a relatively large influence on total performance. However, good speedup results are still achieved (e.g., 29.3 for 32 processors). This is a case of intermediate performance of our code.

As the third example, we choose two protein molecules, HIV protease (1563 atoms and 99 subsystems) and HIV protease dimer (3126 atoms and 198 subsystems). These molecules have three dimensional structures, so the local communication behavior is not as good as for linear molecules. There is also a relatively large variation in the size of the subsystems, which induces more load imbalance problems. This represents the worst case for the performance of our code. Using 32 processors, we obtain a speedup of 19.0 for HIV protease and 24.7 for its dimer. Therefore, in our code, the influence of communication and load imbalance on performance is only moderate, even for this case. From Figure 2, it is noted that larger sized molecules have better overall speedup performance. Because the current code is aimed at dealing with very large molecules, this is encouraging.

We also take some larger molecules as final examples. The HIV protease 4-mer (6252 atoms and 396 subsystems), HIV protease 6-mer (9378 atoms and 594 subsystems), and HIV protease 8-mer (12,504 atoms and 792 subsystems) molecules are chosen. For these large molecules, the memory of one processor cannot afford to carry out the calculations with just one processor. Here we can take advantage of our memory distribution scheme. We employ 2, 8, and 16 as starting numbers of processors to calculate these molecules, respectively. The CPU time and relative speedups compared with the starting number of processors are listed in Table II. Impressive speedup data are obtained. For the largest systems of HIV protease 6-mer and 8-mer, nearly perfect speedup is observed.

To examine the overall performance of our code for treating very large molecules, we performed calculations of a four-iteration SCF energy calculation followed by the calculation of all energy gradients for the HIV protease 8-mer (12,504 atoms). (All other calculations just described were performed on 300-MHz T3E processors; calculations described in this section were performed on an upgraded T3E system with 450-MHz processors.) Table III lists the wall times for different parts of

TABLE III.
Wall Times for Different Parts of Calculation for HIV Protease 8-mer (12,504 Atoms) Using Different Numbers of Processors.[a]

PEs	Four-iteration SCF	Energy gradient	Other parts	Total
12	1607.40	2018.14	197.65	3823.19
16	1216.99	1527.96	165.17	2910.12
24	827.18	1030.83	132.55	1990.56
32	631.39	773.05	115.91	1520.35

[a]Data are expressed in seconds.

the calculations with 12, 16, 24, and 32 processors. The relative speedup (to 12 processors) is plotted in Figure 3. Near perfect speedup is observed. The results show that geometry optimization is attainable for molecules of this size on currently operating parallel machines.

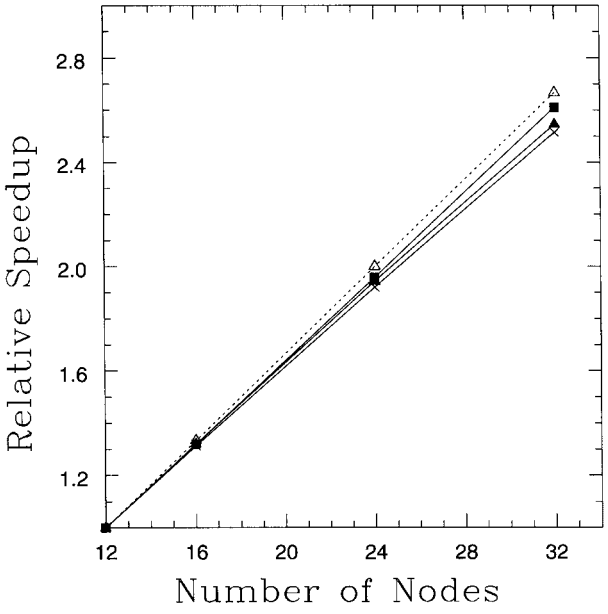

FIGURE 3. Relative speedup (to 12 processors) versus the number of processors of the different parts of calculation for HIV protease 8-mer (12,504 atoms and 799 subsystems). Open triangle: the perfect speedup; solid triangle: speedup for four-iteration SCF calculations; solid square: speedup for energy gradient calculations; cross: speedup for total wall time. The calculations were done on a Cray T3E with 450-MHz processors.

Concluding Remarks

We have implemented a parallel divide-and-conquer semiempirical quantum mechanical calculation code. There are two key features in this code. First, we have implemented a very efficient memory distribution scheme. This scheme, combined with the sparse matrix storage method, greatly reduced the demanding requirement of memory for very large molecular systems. Second, we have used the linear scaling divide-and-conquer approach to achieve the parallel diagonalization of the Fock matrix. The major communication tasks in our code have local behavior and are only a small part of total computing task. The better performance for larger molecules is a very encouraging result for applications to very large molecules. Developmental work is in progress to further improve the load imbalance problem and to construct a suitable dynamic load balance approach.

Acknowledgments

The North Carolina Supercomputing Center provided a CPU time grant for this research.

References

1. W. Yang, *Phys. Rev. Lett.*, **66**, 1438 (1991).
2. W. Yang, *Phys. Rev. A*, **44**, 7823 (1991).
3. W. Yang and T-S. Lee, *J. Chem. Phys.*, **163**, 5674 (1995).
4. G. Galli and M. Parrinello, *Phys. Rev. Lett.*, **69**, 3547 (1992).
5. X.-P. Li, R. W. Nunes, and D. Vanderbilt, *Phys. Rev. B*, **47**, 10891 (1993).
6. F. Mauri, G. Galli, and R. Car, *Phys. Rev. B*, **47**, 9973 (1993).
7. P. Ordejón, D. Drabold, M. Grumbach, and R. M. Martin, *Phys. Rev. B*, **48**, 14646 (1993).
8. E. B. Stechel, A. R. Williams, and P. J. Feibelman, *Phys. Rev. B*, **49**, 10088 (1994).
9. S. Goedecker and L. Colombo, *Phys. Rev. Lett.*, **73**, 122 (1994).
10. T-S. Lee, D. York, and W. Yang, *J. Chem. Phys.*, **105**, 2744 (1996).
11. S. L. Dixon and K. M. Merz Jr., *J. Chem. Phys.*, **104**, 6643 (1996).
12. J. J. P. Stewart, *Int. J. Quantum Chem.*, **58**, 133 (1996).
13. A. D. Daniels, J. M. Willam, and G. E. Scuseria, *J. Chem. Phys.*, **107**, 425 (1997).
14. W. Thiel and D. G. Green, In *Methods and Techniques in Computational Chemistry*, E. Clementi and G. Corongiu, Eds., STEF, Cagliari, 1995, p. 141.

15. R. A. Kendall, R. J. Harrison, R. J. Littlefield, and M. F. Guest, *Reviews in Computational Chemistry*, Vol. 6, VCH, New York, 1995, p. 209.
16. A. M. Marquez, J. Oviedo, and J. F. Sanz, *J. Comput. Chem.*, **18**, 159 (1997).
17. T. R. Furlani and H. F. King, *J. Comput. Chem.*, **16**, 91 (1995).
18. J. J. Vincent, J. Kenneth, and M. Merz, *J. Comput. Chem.*, **16**, 1420 (1995).
19. M. Snir et al., *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996.
20. R. S. Mulliken, *J. Chem. Phys.*, **36**, 3428 (1962).
21. J. J. P. Stewart, *J. Comput. Chem.*, **10**, 209 (1989).
22. J. J. P. Stewart, *J. Comput. Chem.*, **10**, 221 (1989).
23. Q. Zhao and W. Yang, *J. Chem. Phys.*, **102**, 9598 (1995).
24. J. J. P. Stewart, *J. Comput.-Aid. Molec. Des.*, **4**, 1 (1990).